PROGRAMMING PLATFORM FOR DISTRIBUTED ROBOTICS: PRIMITIVES AND PORTABILITY

BY

YIXIAO LIN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Associate Professor Sayan Mitra

ABSTRACT

The Stabilizing Robotics Language (StarL) programming framework aims to simplify development of distributed robotic applications by providing programming abstractions and building blocks for communication, motion control and coordination between robots. It has been used to develop applications such as formation control, automatic intersection protocol, and distributed collaborative search. In this thesis, we introduce the programming abstractions as StarL primitives that are platform independent and useful across hardware platforms, resulting in portability. We first introduce the primitives as building blocks to easily develop, simulate and debug distributed robotic applications in StarL. Then, we discuss the design of the StarL framework which enables us to achieve portability of robot programs across hardware platforms. Thus, the same application program, say, for formation control, can now be ported and deployed on multiple, heterogeneous robotic platforms. We evaluate the design of these new features by simulating several applications.

To my parents, for their sacrifice and unconditional love.
To Xiaohang, for her unending support.

ACKNOWLEDGMENTS

This research is made possible by many individuals. Professor Sayan Mitra provided guidelines and made many significant contributions throughout this research project. Thanks to Adam Zimmerman who laid important foundations of StarL. Special thanks to Professor Taylor Johnson and his student Nathan Hervey for extending the StarL framework to new hardware platforms. Thanks to Professor Nitin Vaidya and the members of my research group, Ritwika Ghosh, Zhenqi Huang, Shuting Li, Liyi Sun, Kenji Fukuda and others, for valuable discussions, implementing and using the StarL framework. I also thank the National Science Foundation for partly supporting this work through research grant (NSF CAREER 1054247).

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

StarL	Stabilizing Robotics Language
gvh	Global Variable Holder
MW	multi-writer multi-reader shared variable
SW	single-writer multi-reader shared variable
ICP	Intersection Coordination Protocol

CHAPTER 1

INTRODUCTION

Distributed robotic systems enable distributed coordination, data replication, and reliability through redundancy. As a result, they are at the frontier of manufacturing [1, 2], transportation [3], logistics [4, 5], exploration [6], environmental monitoring [7], and other engineering systems. While distributed robotic systems have many potential advantages, in realizing their full potential, we have to overcome several technical challenges related to programming and debugging distributed robotic applications. StarL is being developed to help programmers better manage the robot's mobility, communication between robots and distributed coordination [8, 9, 10, 11, 12]. In addition, as part of the work presented in this thesis, StarL now enables code portability, which is essential to deploy distributed robotic applications to different distributed hardware platforms.[1]

1.1 Challenges in Developing Distributed Robotic Applications

Many challenges arise when developing distributed robotic applications. Unlike a conventional computer program, a robot is an open system and has to interact with the highly uncertain physical environment. A robot has the ability to move by controlling its actuators, and it can make sense of the environments by reading values from its sensors. In addition, robots need to communicate with other robots through messages over wireless channels, which allow more complex tasks to be accomplished through coordination. In practice, these channels lead to message delays and drops.

Consider developing a line formation application for a swarm of robots.

[1]The StarL framework is open source and available at `https://github.com/lin187/StarL1.5/`.

The robots start at arbitrary locations and they should form a line with equal spacing between them. For this relatively simple task, robots need to exchange messages over a wireless network about the relative positions of each other and then decide where to form the line. They also need to plan their paths avoiding each other and obstacles in a shared physical space. Then they should control the actuators to move alone the planned path. The interaction of the subroutines handling each of these different subtasks can quickly become overwhelming.

Developing applications in a distributed robotic system is already difficult, porting these applications to a different distributed robotic system is often as hard as re-writing the application. After the line formation application described above has been built for one platform, say a collection of iRobot Create ground vehicles, porting to a different platform, say ARDrone quadcopters, will require a huge amount of work. It might be even easier to develop the application again from scratch.

The design goal for the StarL framework is to make developing distributed robotic applications faster, easier, more abstract, and closer to high-level textbook pseudo-code [13, 14]. The result would be applications that are easier to debug and to port to different hardwares.

1.2 Related Work

Research in distributed robotic systems such as multi-robot systems and swarm robots have demonstrated the benefits as well as the limitations of the distributed approach [15, 16, 17, 18]. Several projects have pushed the limits of custom-built software and hardware for distributed robotics (see [19, 20, 21] for some recent examples); this approach is orthogonal to our objective of developing modular, reusable, and portable distributed robotic applications.

There are several libraries for programming robots using languages like Python and Matlab [22, 23, 24, 25, 26, 27, 28, 29, 30] and notably the Reactive Model-based Programming Language (RMPL) [31], but they do not provide high-level coordination and control APIs nor do they address portability.

The Robotics Operating System (ROS) [22] provides a library of drivers and functions for programming robots and has an ever-growing community of developers and users. While parts of ROS library are reusable, there is

currently no effort towards automatically porting programs across platforms.

An earlier version of the StarL framework was developed by Adam Zimmerman [8], and it consisted of a collection of StarL primitives (e.g. LearderElection, Broadcast and etc.), their implementations, and several applications built using those StarL primitives. This earlier version was also used in [9] to develop a technique for debugging distributed robotic applications using their runtime traces. The StarL high-level language is being developed [11]. The StarL high-level language is used in this thesis to present algorithms, however, it will not be discussed in detail.

1.3 Thesis Contributions and Overview

This work builds upon the previous framework and provides (1) a general-purpose StarL primitive ReachAvoid for motion control, (2) the Distributed Shared Memory primitive for implicit communication, (3) primitives for mutual exclusion (MutualExclusion) and set consensus (Registration), (4) a new software architecture for supporting portability and heterogeneous robots, and (5) demonstrates the effectiveness of the above features through the development of several new applications. The applications include intersection coordination protocol, distributed search and formation. Parts of the work presented in this thesis appeared in other papers [10, 11, 12].

The StarL (Stabilizing Robotics Language) framework offers a collection of high-level functions, referred as *StarL primitives*, aimed to help programmers develop distributed robotic applications. Applications written using the StarL primitives can be ported to different distributed robotic hardware platforms with much less effort than what is possible with existing systems.

StarL primitives are provided to help programmers better manage robot's mobility, communications and distributed coordination. The StarL primitive for the robot's motion is provided. In addition, StarL primitives for communication include both message passing and distributed shared memory. StarL primitives also include implementations of many useful distributed algorithms such as leader election, mutual exclusion and set consensus, and it can be extended to support any distributed algorithms.

The StarL framework provides the infrastructure necessary for supporting different hardware platforms. Instead of building everything from scratch,

only the low-level functions that interface with StarL need to be changed while everything else remains the same. StarL also provides a simulator with the capability of simulating the application with heterogeneous robots. StarL helps a programmer develop, simulate and debug distributed robotic applications on one or more types of robotic hardware platforms.

1.4 Organization

In Chapter 2, we will first introduce the software components of StarL. Then we will show how to create a simple race application using the StarL primitives. Next, in Chapter 3, an extensive discussion about the StarL primitives that addresses the motion, communications, and coordination will be presented. Then, software architecture, programming abstractions, portability and heterogeneous robots are discussed in Chapter 4. In Chapter 5, StarL applications including Intersection Coordination, Distributed Search and Formation are presented. Finally, we discuss the future directions and conclusions in Chapter 6.

To learn how to create distributed robotic applications in StarL, read Chapter 2. Chapter 3 will be useful to take advantage of pre-built high-level functions and for others to develop their own high-level functions. Chapter 5 also provides guidelines to create and simulate your own distributed robotic applications. Chapter 4 is useful for supporting StarL on different hardware platforms and creating new programming abstractions for robots.

CHAPTER 2

PROGRAMMING IN STARL

In this chapter, we will first briefly introduce the software components of StarL. Then a simple example application will be presented to show how to write a StarL application.[1]

2.1 Overview of Software Components in StarL

StarL is composed of three main software components: the StarL library, the StarL applications, and the StarL simulator. The StarL library offers StarL-specific Java classes, interfaces layer functions, and StarL primitives. In a StarL application, programmers use functions and primitives from the StarL library to write distributed robotic applications for robots to accomplish complex tasks. Finally, the StarL simulator is a discrete even simulator which can simulate several instances of a StarL application running on multiple robots. This is a useful tool for testing and debugging applications.

2.1.1 StarL Library

The StarL library consists of the following collection of Java files:

- StarL-specific Java classes are classes used extensively when programming in StarL. For example, the ItemPosition.java is the class for representing the coordinates of a way-point in $\mathbb{R}^3$. LogicThread.java is the class for using threads in StarL.
- A robot model defines the robot's state, sensors, dynamics, and available motion commands.

[1] This chapter includes previously published material [12].

- Interface layer functions, including the interfaces and implementations, are hardware-independent functions that achieve some low-level tasks. For example, the getMyPosition function gets the robot's current position.

- *StarL primitives*, including the interfaces and implementations, are high-level functions that achieve some high-level tasks while abstracting away the low-level details. For example, ReachAvoid primitive can be used to reach the designated target while avoiding the specified unsafe regions.

For each robot, many threads are needed to support the functions such as message passing, sensor value streaming, motion controlling and the running the StarL application. StarL creates an interface, the global variable holder (*gvh*), such that each thread (LogicThread) writes to variables related to its function while each thread can read from all the variables in the *gvh*. For example, the thread for the positioning system updates the robot's current position while the robot's motion controller thread reads the current position to determine the next motion command.

The interface layer functions StarL library provide have lower levels of abstraction than StarL primitives. For example, Unicast is a function while Broadcast is a primitive. The interface layer and the StarL software architecture is discussed in Section 4.1. StarL primitives include motion primitive, distributed algorithms primitives and communication primitives. Each category addresses some issues when programming for distributed robotic systems. StarL primitives are discussed in detail in Chapter 3.

2.1.2 StarL Application

A StarL application is a distributed robotic application that accomplishes some tasks. The StarL application is defined by a StarL thread that runs the algorithms to accomplish the application tasks. This thread will be started by the *gvh* with some initialization, followed by a while loop. Inside this while loop, a state machine needs to be defined by the programmer. In this state machine, the algorithms for the application tasks are implemented using functions defined by the StarL interfaces and StarL primitives.

2.1.3 StarL Simulator

A discrete even simulator is provided so that the written StarL application can be visualized and debugged easily. StarL applications can also be deployed to actual robotic hardwares, more details can be found in Section 4.2.

The StarL simulator runs the same application with user specified number of robot instances in a simulated physical environment with detailed physics models. The StarL simulator allows a developer to run an application under a broad range of conditions. A large set of simulation parameters can be tuned including message delays, message loss rate, obstacles in the physical environment, crash failures, and so on. These parameters are specified in another Java file (Main.java).

Different types of robot models can also be provided to the simulator to simulate different robots. A robot model includes the robot's state, sensors, and interfaces to the motion commands as well as the motion dynamics. In the current implementation, Models of the ground robot (iRobot Create) and the quadcopter (ARDrone 2) are provided.

2.2 Preview of a StarL Application: Race

StarL comes with a few applications. Take the Race application as an example, in which a number of robots are given a common sequence of way-points as input. Starting with the first point in the sequence, each robot moves to the target point using the *ReachAvoid* primitive. When any robot has reached the vicinity of the target point, it informs other robots using the *Broadcast* primitive, before moving on to the next way-point in the sequence. Upon getting the message, the robot changes the target to the next point in the sequence. The pseudo code for the Race application is shown in Figure 2.1.

The pseudo code is written in the StarL high-level language. The StarL high-level language describes a set of *actions* in the usual guarded-command style. An action *may* occur (is enabled) when the **pre**condition (or guard) is true, and when it *does* occur, the state is then changed according to the statements in the **eff**ect. In the actual Java implementation, these actions correspond to if conditionals inside a while loop. An application thread is started and initialized at the beginning. Then, in each iteration of the

```
1  loc: enum{init, pick, wait} = init;
2  target[ ] : ItemPositon
3  currentIndex: int;
4
5  initialize()
6    pre loc == init
7    eff target = getInput(); currentIndex = 0; loc = pick;
8  pick()
9    pre currentIndex < targets.size() && (loc ==
         pick||!activeFlag||failFlag)
10   eff doReachAvoid(target[currentIndex],getObstacles()); loc = wait;
11 wait()
12   pre loc == wait && doneFlag
13   eff currentIndex ++; loc = pick; Broadcast(currentIndex)
14
15 receivedMessage(m)
16   currentIndex = m.currentIndex + 1; loc = pick;
```

Figure 2.1: Pseudo code of the Race application

while loop, each if condition is checked. When the if condition is met, the corresponding code block is executed.

There are three states: init, pick and wait. The *ItemPosition* is a built-in class type for storing 3D coordinates of points in space with respect to a common and fixed coordinate system. As shown in line 9 of Figure 2.1, when there is at least one point to be reached, the *ReachAvoid* motion primitive is used to reach the target point while avoiding the unsafe set (static obstacles in the environments in this application). The *ReachAvoid* motion primitive provides three flags to inform the application program about the status of motion. The *activeFlag* indicates that the *ReachAvoid* primitive is in progress; the *doneFlag* is raised when the robot has been in the vicinity of the target point; and the *failFlag* is raised when the robot has entered the unsafe set. The *ReachAvoid* is explained in detail in Section 3.2. This application highlights the usage of two StarL primitives: *ReachAvoid* and *Broadcast*. The StarL primitives are building blocks for more complex tasks and they greatly simplify the code for the StarL application program. The specifications, usage and implementations of StarL primitives are discussed in detail in Chapter 3.

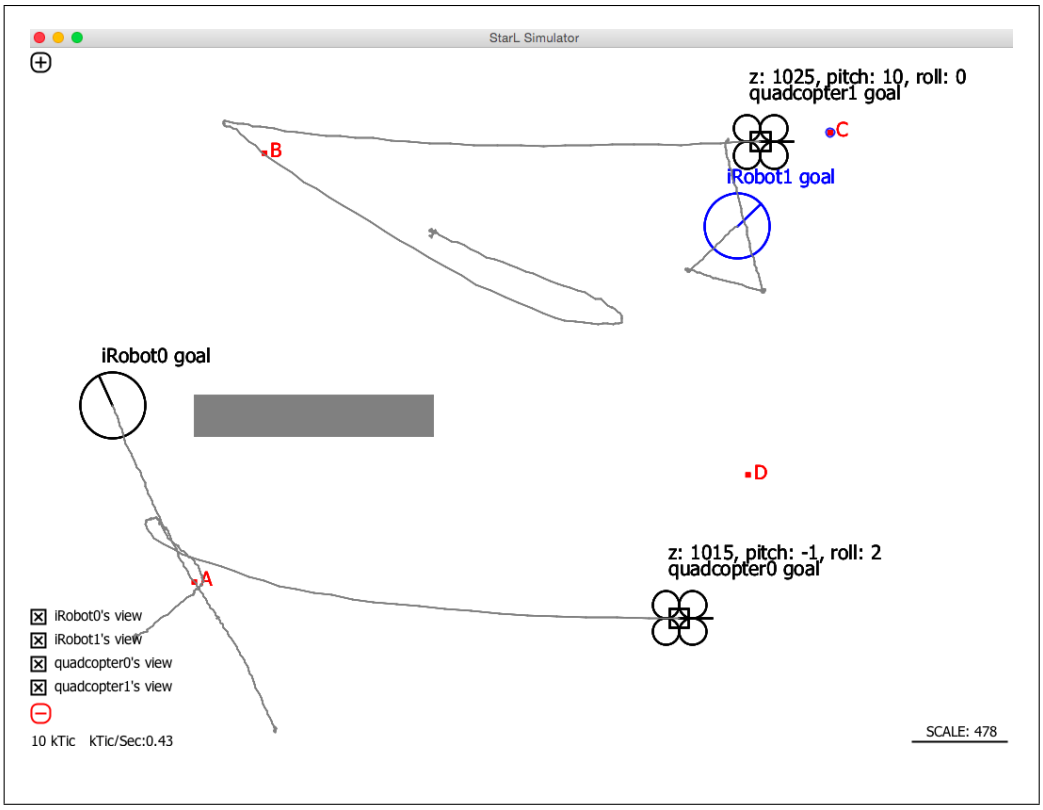

Figure 2.2: Screenshots of the Race application simulation

The screen shots of a Race application simulation is shown in Figure 2.2. In the simulation, each robot is shown alone with the robot's state information and its movement trace. There are four destinations points in the sequence A, B, C and D, shown in red circles. With the obstacles shown in the gray box on the left, two iRobots and two quadcopters are given the sequence A, B, C and D. The iRobots and the quadcopter have very different dynamics. For example, the quadcopters need to takeoff before moving toward a destination point. The quadcopters also move much faster than the iRobots. At the same time, the quadcopters overshoot more than the iRobots. Motion of heterogeneous robots are discussed in Section 4.3. The sudden change of motion direction shown in the traces are the results of the robots being informed that some other robot has reached the target point. Therefore the robots change direction and move on to the next point in the sequence.

CHAPTER 3

STARL PRIMITIVES

Deterministic abstractions are easier to understand and reason about than nondeterministic ones. Programs for a distributed robotics system, however, have to deal with nondeterminism arising from uncertainties in communication, dynamics, and failures. We make the choice of carefully exposing some of these nondeterministic factors to the programmer through the StarL primitives. We ameliorate the loss of determinism by making the primitives *uniform* in the following way: Each StarL primitive is defined by a specification which includes an input/output interface and certain precisely stated properties.[1]

3.1 Specifications of StarL Primitives

A StarL application interacts with a StarL primitive through the global variable holder (*gvh*). Recall that the *gvh* is an interface created for multithreading, such that each thread writes to variables related to its function while each thread can read from all the variables in the *gvh*. First, the primitive is invoked by its name by the application just like a regular function call, and then the progress and results of the call can be checked by reading specific variables.

For example, the StarL *Mutex* primitive implements a distributed mutual exclusion algorithm that allows a fixed set of processes to access a set of shared resources in a mutually exclusive fashion. The set of participating processes is specified by a list called *pList* and the set of shared resources is specified by a set called *sharedResources*. Each robot's request is stored in a set-valued variable called *myreq*, which could be any subset of the *sharedResources*. A StarL application uses this primitive as shown in

[1]This chapter includes previously published material [10, 11].

Figure 3.1.

```
1    mux = Mutex(id, pList);
2    myreq = // a subset of the sharedResources
3
4    init()
5      pre true
6      eff mux.do_mutex(myreq)
7
8    use()
9      pre mux.crit(myreq)
10     eff // use myreq
11     mux.release(myreq)
12
13   retry()
14     pre mux.failed(myreq)
15     eff // handle failure
```

Figure 3.1: Usage of the Mutex primitive

The pseudo code is written in the StarL high-level language. Recall that the StarL high-level language describes a set of *actions* in the usual guarded-command style. An action *may* occur (is enabled) when the **pre**condition (or guard) is true, and when it *does* occur, the state is then changed according to the statements in the **eff**ect.

The first line creates the mutual exclusion object with a globally unique identifier *id* and with the set of participating processes *pList*. By default, the *sharedResources* is the set of all possible string values.

The variables *mux.crit* and *mux.failed* in process i's *gvh* are written by the mutual exclusion algorithm to indicate whether i has obtained access to the requested set *myreq*, or whether the mutex algorithm has failed. In this example, the *init* action makes a request for mutual exclusion, and the *use* action is only enabled when this access is granted by the underlying implementation of the mutual exclusion algorithm. Thus, if multiple processes execute this application concurrently, then only one of the processes can execute the *use* action at a time.

Each primitive has a hardware-independent, abstract, and nondeterministic specification as well as an implementation in Java. The motion-related primitives also have hardware platform-specific implementations. Under rea-

sonable assumptions about the environment, the implementations meet the specifications.

The primitives are implemented using interface layer functions and other primitives. Recall that the interface layer functions are hardware-independent functions that achieve some low-level tasks. The interface layer and the StarL software architecture is discussed in Section 4.1. If the underlying assumptions are violated, some properties of the primitive may be violated. Under such circumstances, the primitive should raise a failed flag to the StarL application. Developers can handle this *exception* using best-effort strategies.

StarL primitives are divided into three categories: motion, communications and distributed algorithms. In the following section, we enumerate the specifications and explain the provided implementations.

3.2 Primitive for Motion

Motion is an essential part of any mobile robot. The *ReachAvoid* primitive generalizes the motion of mobile robots by allowing the application to specify a $target$ point and an $unsafe$ region such that the robot will try to reach the target point while avoiding the unsafe region. The *ReachAvoid* also provides three motion flags to inform the application about the progress.

3.2.1 Specification

The interface includes the following:

- $doReachAvoid(target, unsafe)$ function specifies the $target$ and the $unsafe$ region and instructs the underlying motion controllers to start the primitive.
- $doneFlag$ is a Boolean variable that is set to true when the robot has reached the vicinity of the target.
- $failFlag$ is a Boolean variable that is set to true when the robot has entered the unsafe region. This could happen due to uncertainties in the environment. For example, a sudden gust of wind could blow the quadcopter to the $unsafe$ region regardless of the controller's effort.

- *activeFlag* is a Boolean variable that is set to true when the *ReachAvoid* primitive is active and is set to false otherwise. The primitive is implemented using best-effort strategy. After invoking the primitive, if all the flags are set to false, this means that the *ReachAvoid* primitive has given up. This could happen if no safe path to the target is found by the path planning algorithm or if some unknown obstacle is blocking the robot.

The properties of the *ReachAvoid* primitive include the following:

- (safety) The position of the robot is always outside the region in *unsafe*.
- (goal) If the *doneFlag* is set to true, then the position of the robot has been within the vicinity of the *target* in the recent past. The vicinity is a tunable parameter. It should be tuned based on the accuracy of the positioning system, the precision of the motion automation and the requirements of the task.
- (progress) The *activeFlag* is set to false in a finite amount of time.

The *MotionControl* properties are expected to hold under the following assumptions about the execution environment and the low-level controllers. The actuators do not fail; the accuracy of the positioning system is at least vicinity/2; outside the region in *unsafe*, there exists a path between the robot and the target, where the width of the path is greater than some constant d. The value of d is related to the accuracy of the low-level motion controller, the positioning noise, and the movement pattern of the robot.

3.2.2 Implementation

The *ReachAvoid* primitive is implemented using two parts: the RobotMotion interface provided by the interface layer and a rapid exploring tree (RRT) path planning algorithm [32]. The RobotMotion is a way-point tracking function. Given a target point and the robot's current position, the RobotMotion interface uses the motion commands available on the robotic platform to control the robot to move to the target point. The motion commands are dependent on the robot's hardware platform. The RobotMotion interface is discussed in detail in Section 4.3.

Given the *start* position, the *target* position, the *unsafe* region and the safe radius from the center of the robot to any unsafe region, the RRT instantiates a tree T in $\mathbb{R}^3$ using the *start* as the root. Then, the RRT algorithm adds new points to the tree as follows: It picks a random point $\vec{x} \in \mathbb{R}^3$ and attempts to add it to T. In doing so, it first finds the point in the tree $\vec{p} \in T$ that is closest to $\vec{x}$. Then it simulates the robot's movement from $\vec{p}$ to $\vec{x}$ using the RobotMotion interface and the robot's physics model. Finally, if the simulated path is sufficiently far from *unsafe*, then $\vec{x}$ is added to tree T. Otherwise, another point halfway between $\vec{p}$ and $\vec{x}$ is picked unless the distance between the two falls below a threshold in which case a new $\vec{x}$ is picked at random. After every new node addition, the algorithm tries to add the *target* to tree T following the same procedure.

The tree construction stops if either the *target* has been added to the tree or the size of the tree T reaches a threshold size. If a path from the vicinity of *start* to the vicinity of *target* exists in T then this path (a list of points) is sent to the RobotMotion interface. Otherwise, the *activeFlag* is set to false. Assuming that *start* and *target* did not change during the tree construction, this indicates to the StarL application that a safe path has not been discovered by the path planning algorithm. In general, establishing that a safe path does not exist is challenging, and our design of the runtime system leaves it to the programmer to code best-effort strategies by detecting when all the control flags (*activeFlag*, *doneFlag*, and *failFlag*) simultaneously become *false*.

After the RRT has generated a path, the first point in the path is fed into the RobotMotion interface, upon reaching its vicinity, the next point in the list is fed in. Eventually, the robot will reach the vicinity of the target point and the *doneFlag* is set to true. The usage of the *ReachAvoid* primitive and the interaction between the application and the motion flags are shown in Figure 2.1.

3.3 Primitives for Communication

Communication between robots provides the foundation for distributed algorithms by allowing robots to coordinate when facing complicated tasks. Distributed algorithms can be either based on message passing or on distributed

shared memory. Accordingly, the StarL communication primitives include the *Broadcast* and the *Geocast* primitives and the *DistributedSharedMemory* primitive.

3.3.1 Broadcast and Geocast

The *Broadcast* and the *Geocast* primitives are implemented using the send (unicast) method, the receive method and the communication protocol from the interface layer. The *Geocast* primitive also uses the robots' location information that is accessible through the *gvh*. The underlaying communication protocol is a simple protocol that asks for acknowledgment for each messages being sent. This can be easily expanded to enforce message ordering, such as FIFO ordering, causal ordering, and total ordering.

The interface includes the following:

- *broadcast*(m) is a function used to send out message m to every other robot in the network. All processes will receive m within some constant d_1 time of the broadcast. The constant d_1 is a hardware platform-specific parameter, where the value of d_1 depends on the delay and reliability of the communication channels.

- *geocast*(m, A) is a function used to send out message m to robots that are in the area A.

The properties of the *Geocast* primitive include the following. If a message m is sent through the *geocast* function at time t_0 then within some constant d_2 time of the *geocast*, the following properties hold:

1. (exclusion): Any process continuously located outside A during the time interval $[t_0, t_0 + d_2]$ will not receive m.

2. (inclusion): Any process located within A during the time $[t_0, t_0 + d_2]$ will receive m within the time interval $[t_0, t_0 + d_2]$.

The constant d_2 is a hardware platform-specific parameter, where the value of d_2 depends on the delay and the reliability of the communication channels. For a robot moving in or out of A during $[t_0, t_0 + d_2]$, the message may or may not be delivered; but a robot outside A is guaranteed not to receive the message. The inclusion property can only be guaranteed under additional

assumptions about messages being delivered in a bounded amount of time within the area A.

Implementation of the *Geocast* primitive over a wireless network involves details like tagging the message with the location of the originating process, resending messages in the absence of acknowledgments, and dropping certain messages based on the receiver's location.

3.3.2 Distributed Shared Memory

The *DistributedSharedMemory* primitive in StarL allows developers to easily implement many distributed algorithms [33, 34, 35]. Without this primitive, shared memory based distributed algorithms must be converted to use message passing, which is a time-consuming and error-prone process.

3.3.2.1 Specification

Each process could use multi-writer multi-reader (MW) shared variables and single-writer multi-reader (SW) variable arrays. The shared variables do not need to be declared explicitly. The corresponding variable will be created when the first write to the variable is executed. The reads and writes are implemented as a StarL primitive using the message interface. The interface includes the following:

- *create* is the declaration statement for initializing an MW shared variable.
- *get* is a function used to read the value of the shared variable.
- *put* is a function used to update the value in the shared variable.

The distributed shared memory can have many different underlying consistency models. The one that is currently implemented is eventual consistency [36].

- (eventual consistency): If no $put(\text{“}x\text{”}, Val')$ is performed after $put(\text{“}x\text{”}, Val)$, then eventually, all subsequent $get(\text{“}x\text{”})$ that are performed by any process will return Val.

Eventual consistency is guaranteed if messages are delivered in a finite amount of time and the participating processes do not fail.

3.3.2.2 Implementation

MW variables can be written to by different processes. Therefore, mutual exclusion should be used to prevent processes from writing to the same variable at the same time. If two processes write to the same MW variable, the value with the latest timestamp will be populated eventually. The timestamp does not need to be specified explicitly, a new timestamp is obtained when the *put* function is executed. To initialize a MW variable properly, the *create* function should used. The *create* function declares a MW shared variable with the initialized value and then specify the timestamp as -1.

Unlike MW variables, the SW variables always come as arrays. When writing to the SW variable array x, $put(\text{“}x\text{”}, Val')$ will figure out which index in the x array belongs to the current process and perform the write. Reading the variable must specify the owner of the variable it wants to read from: $get(\text{“}x\text{”}, owner)$.

Each shared variable is associated with a name, owner, and some attributes that are key value pairs. For example, a shared variable can have name = “position”, owner = “robot1”, Key1 = “x”, value = 2, Key2 = “y”, value = 1, Key3 = “z”, value = 5. If no key is given during the *put* or *get*, the “default” value will be updated or read. The shared memory primitive is implemented such that any object that can be serialized to string is a valid object type in the shared memory.

When *put* is invoked, a timestamp is obtained and the corresponding value in the local copy will be updated. Then the value, along with the timestamp are propagated to other processes using the *Broadcast* primitive. Upon receiving the others' messages, the local copy of that variable is updated.

When *get* is invoked, the local copy of the variable will be returned. For the SW variable array, when reading variables that are owned by other processes, or for an MW variable that is not initialized, it is possible that the write has not propagated to the reading process through message passing. In that case, a null object will be returned. In that case, the programmer must check null before parsing the returned value.

3.4 Primitives for Distributed Algorithms

3.4.1 Registration

The *Registration* primitive solves a set-valued distributed consensus problem where a set of processes agree on the identities of the participants. The interface includes the following:

- *Register* is a function used to create a registration object.
- *do_register* is a function used to start the registration process.
- *undo_register* is a function used for a process to leave the registration.
- $\langle rList, ts \rangle$ is a pair of variables. The *rList* is the agreed set and *ts* is the timestamp indicating when *rList* was computed. Otherwise *rList* stores a *null* value.

Suppose processes $p_1, p_2, ..., p_k$ invoke *do_register* at time $t_1, t_2, ..., t_k$, then the properties of the *Registration* primitive include the following:

- (agreement) For any two processes p_i and p_j with agreement timestamps (*ts*) within d of each other, the corresponding *rList*s are identical.
- (soundness) For any process p_i, p_i is contained in *rList* with timestamp *ts* only if p_i invoked *do_register* at most d_1 time before *ts*.
- (progress) For any process p_i, if p_i invokes *do_register* then within at most d_2 time, the registration completes with p_i, that is, *rList* contains p_i.

The *Registration* is implemented using the *Geocast* primitive. Assumptions are the same as the ones mentioned in the *Geocast* primitive in Section 3.3.

To support multiple independent *Registration* objects inside the same StarL application, each registration object is invoked with an identifier. A registered process may use the *undo_register* function to trigger a restart of the registration among the remaining processes. The *rList* value can be updated with a new timestamp, and during the interim it may be *null*.

3.4.2 Leader Election

The *LeaderElection* primitive elects a leader among the participating processes. The participating processes will learn about either the leader's identity or the failure of the leader election. The interface includes the following:

- *Election* is a function used to create a leader election object; it takes the list of participants as a parameter.
- *do_election* is a function used to start the election.
- *Leader* is a variable that stores the identity of the leader, *null* if the election is in progress, and *fail* if the election fails.

The properties of the *LeaderElection* primitive include the following:

- (agreement) For any two processes i and j that start election within d time of each other, if *Leader* is not *null*, then their *Leader* values are identical.
- (soundness) For any process i, $Leader = i$ at time t only if i invoked *do_election* and this invocation occurred earliest at $t - d_1$.
- (progress) For any process i, if i invokes *do_election* then within at most d_2 time election completes successfully, that is, *Leader* equals a valid identifier.

The progress property is guaranteed when messages are delivered in a finite amount of time and process i does not fail.

Currently, one of the implementations of leader election is based on randomized ballot creation and another implementation is based on a version of the Bully algorithm [37].

3.4.3 Mutual Exclusion

The *MutualExclusion* primitive allows a fixed set of processes to access an object (or a set of objects) in a mutually exclusive fashion. If a process requests multiple objects, then it gains access to all the objects at the same time, but it may release them one at a time.

The interface includes the following:

- $Mutex(id, pList)$ is a function used to create an mutual exclusion object with an identifier id. The variable $pList$ sets the participating processes.
- do_mutex is a function used to request the access to a set of critical sections.
- $release$ is a function for releasing a critical section.
- $crit$ is a Boolean variable to indicate whether access to all requested critical sections has been granted to this process.
- $failed$ is Boolean variable that is set to true if the $MutualExclusion$ primitive has failed.

The properties of the $MutualExclusion$ primitive include the following:

- (safety): For any two processes, the set of critical sections they have access to are disjoint.
- (progress): If there exists a time bound d_1 within which critical sections are released, then there exists a time bound d_2 within which any requesting process gains access to its critical section(s).
- (non-interference): If no process holds the critical sections being requested by i, then i gains access with time d_3 ($d_3 \ll d_2$).

The safety property is always guaranteed. The progress and non-interference properties are guaranteed if messages are delivered in a finite amount of time and the participating processes do not fail.

The current implementation of the $MutualExclusion$ primitive is based on a modification of the Ricart and Agrawala's algorithm [37]. To support multiple independent mutual exclusions inside the same StarL application, each mutual exclusion object is invoked with an identifier. The usage of this primitive was shown in Figure 3.1.

Summary of StarL Primitives In summary, the primitives address problems related to motion, communications and distributed coordination. The primitives provide the programming abstraction using the pre-defined interfaces and expected properties. Uncertainties, implementations and low-level

details are abstracted away. The primitives enable a programmer to reason and create distributed robotics applications at the level of abstraction of the distributed algorithms.

CHAPTER 4

SOFTWARE ARCHITECTURE AND CODE PORTABILITY

4.1 Software Architecture

The StarL framework is organized into five layers as shown in Figure 4.1. The bottom two layers are hardware platform-specific and the top three layers are portable across hardware platforms.[1] Each layer consists of functions for motion, sensing, and messages at the similar level of abstraction. The lower layers serve as building blocks, while higher layers abstract away the details to allow more advanced capabilities.

The lowest layer is the hardware layer. This layer directly depends on the hardware used in a distributed robotic system. Many different hardware systems can be used to deploy StarL. In our lab at the University of Illinois, the hardware components are the Android phones, iRobot chassis, and the OptiTrack indoor positioning system. The communication channels are Bluetooth channels between the Android phones and the iRobot chassis and the Wi-Fi channel between the Android phones. Deploying the StarL framework to other hardware has also been done by professor Taylor Johnson's group using quadcopters, Raspberry Pi and Kinect camera [38].

The platform layer implements the drivers for the basic functions as building blocks for the logic layer. For example, functions in the platform layer act like hardware drivers for sending motion commands to the iRobot Create chassis, for setting up wireless communication, and for reading data from the OptiTrack indoor positioning system. To run StarL on a distributed robotic system, the drivers in the platform layer construct a bridge for the upper layers to interact with the system's hardware. For simulating the applications, the platform layer is implemented as a simulated hardware platform.

The logic layer wraps the low-level methods into high-level methods that

[1] This chapter includes previously published material [10, 11, 12].

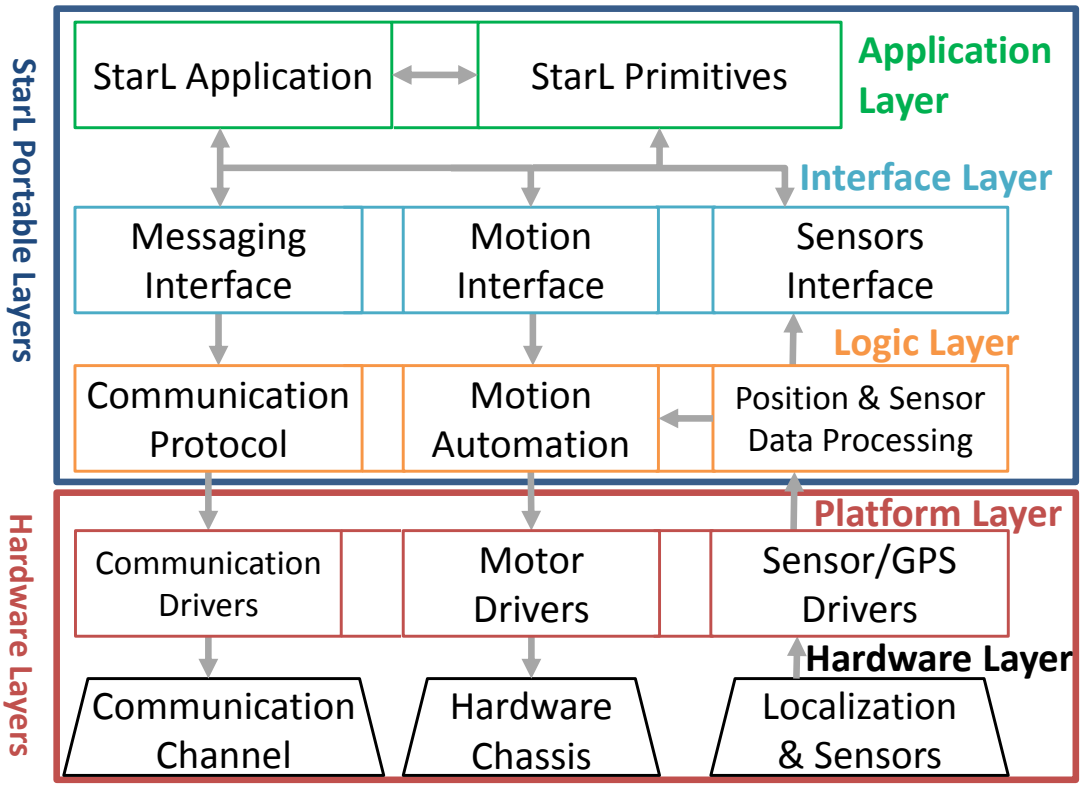

Figure 4.1: StarL architecture

will be provided to construct the interface layer. For example, the interface layer defines a motion automation while the logic layer implements it. The motion automation is a way-point tacking function which the robot is given a point in space and it will attempt to maneuver to that point. The logic layer uses the the positioning data and the sensor information to determine how the robot should move, then it uses the functions that the platform layer provides to send the motion commands to the robot chassis. The interface layer also has the global variable holder (*gvh*). Recall that the *gvh* is an interface created for multi-threading, such that each thread writes to variables related to its function while each thread can read from all the variables in the *gvh*. The *gvh* also provides access to each part of the framework.

The top layer is the application layer. This includes the StarL primitives (Chapter 3) as well as the StarL applications. The StarL primitives are more advanced functions constructed using functions from the interface layer. For example, the *ReachAvoid* primitive (Section 3.2) uses the way-point tracking function from the interface layer and a RRT path planning algorithm to reach the designated target while avoiding the specified unsafe regions. The StarL application uses both interface layer methods and StarL primitives to accomplish more complicated tasks. For example, the Race application shown in Figure 2.1 uses *getInput()* function from the interface layer to get the user specified list of way-points as input. The Race application also uses both the *Broadcast* and the *ReachAvoid* primitives. More StarL applications are presented in Chapter 5.

4.2 Class Hierarchy

Abstraction and modularity is the key to building the StarL software infrastructure. Abstractions of a module hide its implementation details and provide more relevant descriptions about its properties. Each module can be reasoned about, tested, and debugged separately such that a complex task is decoupled into manageable pieces. Abstraction and modularity also enable a programmer to modify the code on one layer while leaving the other layers unaffected. Therefore, the StarL framework is portable to other hardware platforms by providing the low-level implementations that act like device drivers, while leaving the code above the interface layer unchanged. We

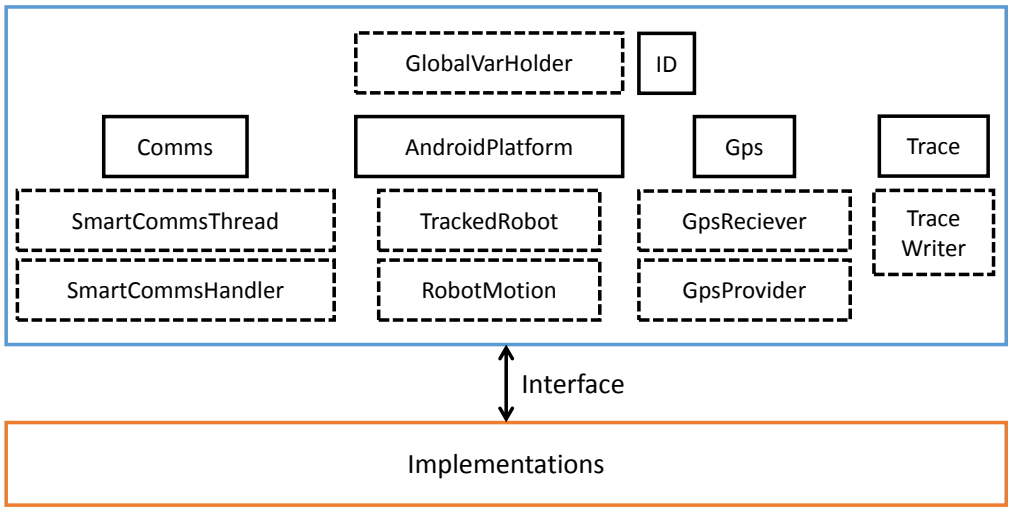

Figure 4.2: Class hierarchy for the gvh

have implemented two different hardware layers: one with iRobot robots, Android phones and the OptiTrack system (mentioned in Section 4.1), the other one with simulated hardware which also includes models of ARDrones. Porting the StarL framework to other hardware has also been done by professor Taylor Johnson's group using quadcopters, Raspberry Pi and Kinect camera [38].[2]

With the abstraction and modularity concepts in mind, interfaces and abstract classes are used extensively in the actual Java implementation.

The global variable holder (*gvh*) is an important part of the interface layer, where communications, robot motion, positioning and sensing are included in this class. The Java classes related to the *gvh* are shown in Figure 4.2. The top blue box includes the files that are in the interface layer and the orange box on the bottom includes the actual implementations. Java interfaces and abstract classes are shown in rectangles with dashed lines. For the abstract classes and interfaces, different implementations can be provided for different hardware platforms. For example, the SimGpsProvider gets the robots' position data using simulated results, where RealGpsProvider gets the robots' position data from the OptiTrack indoor positioning system. Therefore, when porting the StarL to a different hardware platform, only the files below the interface need to be modified while the StarL primitives and the StarL applications stay the same.

[2]demo available at `https://www.youtube.com/watch?v=VQNRM3VvvtQ`.

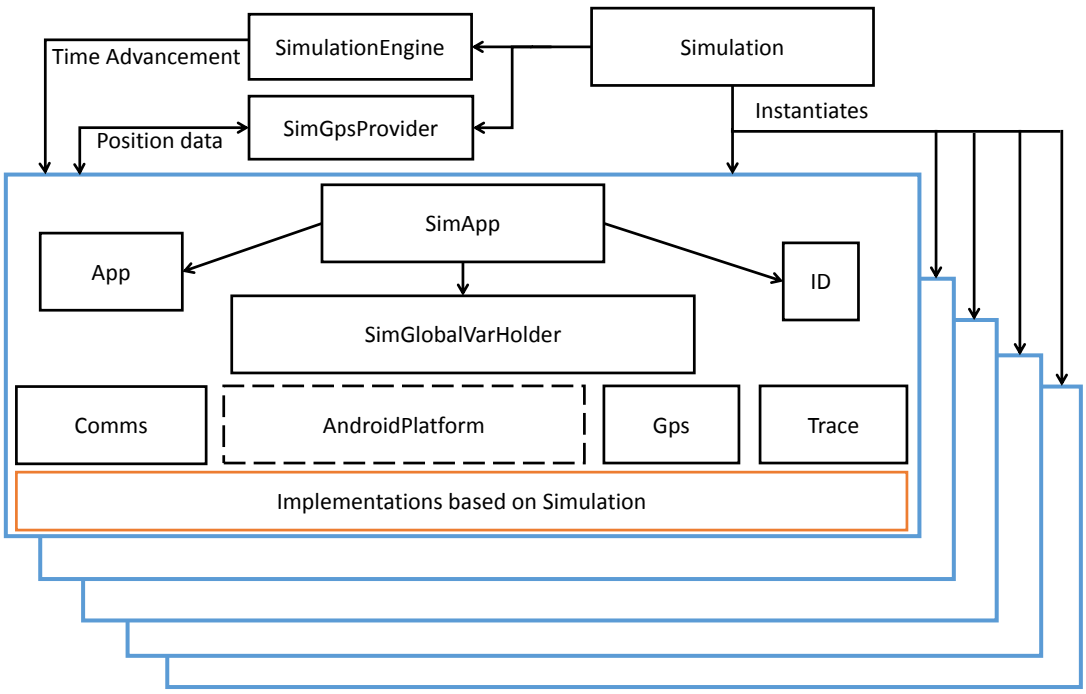

Figure 4.3: Simulator workflow for multiple robots

The RealGlobalVarHolder class will instantiate the implementations that are based on actual hardware, such as SmartUdpComThread, AndroidLogging and UdpGpsReceiver. On the other hand, the SimGlobalVarHolder class will instantiate the implementations that are based on simulations, such as SimSmartCommsThread, SimLogging and SimGpsReceiver.

Figure 4.3 shows the workflow of the StarL simulator when simulating multiple robots. The simulation class starts the SimulationEngine, where the multi-threading and the simulation time advancement are handled. The simulation class also starts the SimGpsProvider, where the robots' positions are calculated and propagated to each robot. The simulation class instantiates multiple copies of the SimApp with different ids to simulate a distributed robotic system. More simulation results of StarL applications will be presented in Chapter 5.

StarL also supports heterogeneous robots. The generalization of motion for heterogeneous robots and the simulation of their physics are discussed in Section 4.3.

4.3 The Motion of Heterogeneous Robots

For some platforms (e.g., an industrial robotic arm vs. a bipedal robot), there is little commonality in the task and their implementations, and therefore, portability is not meaningful. However, there exists a class of distributed robotic platforms and related tasks, such as visiting a sequence of points in 3D space to achieve some higher-level goal, where automatic portability can improve both quality of programs and the productivity of developers. We generalize the motion for this class of robots using the RobotMotion interface.

The RobotMotion interface defines the way-point tracking function. Given a target point and the robot's current position, the robot is controlled properly to move to the target point. The class that implements the RobotMotion uses the available motion commands that are dependent on the robot's hardware platform.

Table 4.1 shows the different available motion commands for the iRobot Create and the ARDrone2. Each shown commands are executed exclusively such that when a new command is issued, the previous command is aborted.

Table 4.1: Motion commands for the ARDrone2 and the iRobot Create

AR Drone2	iRobot Create
takeOff()	straight(v_{ref})
land()	turn(a_{ref})
hover()	curve(v_{ref}, r)
setYawSpeedPitchRollGaz(a_{ref}, θ_{ref}, ϕ_{ref}, v_{ref})	

Recall that within AndroidPlatform (in Figure 4.2), there are two important interfaces: RobotMotion and TrackedRobot. The class that implements the TrackedRobot consists of the dynamical model and the internal states of the robot. It also must provide two functions: predict and update. The input for the predict functions is the time elapsed. Then, the predict function predicts the robot's position using the internal states, the elapsed time and the robot's dynamical model. The simulator then decides whether the robot can move to the predicted point based on whether there are collisions. Then the update function is called to update the robot's position and internal states accordingly. The robot's sensor data should also be stored in the class that implements the TrackedRobot. The dynamical models for the AR Drone2 and the iRobot Create are shown in Table 4.2

Table 4.2: Dynamical models of the AR Drone2 and the iRobot Create

AR Drone2	iRobot Create
$thrust = (gaz + 10)/\cos\phi/\cos\theta$	$\dot{x} = v \cdot \cos\theta$
$\ddot{x} = -(thrust)(\sin\phi \cdot \sin\psi + \cos\phi \cdot \sin\theta \cdot \cos\psi)/M$	$\dot{y} = v \cdot \sin\theta$
$\ddot{y} = (thrust)(\sin\phi \cdot \cos\psi + \cos\phi \cdot \sin\theta \cdot \sin\psi)/M$	$\dot{\theta} = a_{\text{ref}}$
$\dot{z} = gaz$	
$\dot{\psi} = gain \cdot (a_{\text{ref}} - \psi)$	

For the AR Drone2 model, the state variables are x, y, z position coordinates, yaw (ψ), pitch (θ), roll (ϕ) angles and the corresponding velocities. For the iRobot model the state variables are x, y position coordinates and the heading (θ) angle.

Since control commands are provided by the robots' hardware platforms as black boxes, in creating the model for these controller we use standard proportional controllers. For example, the yaw speed control is shown in the last row and similar controls for pitch and roll are used (omitted from the table). This technique can be extended to other closed-loop dynamics such as PID. The RobotMotion reads sensor inputs and uses the motion commands in a closed-loop.

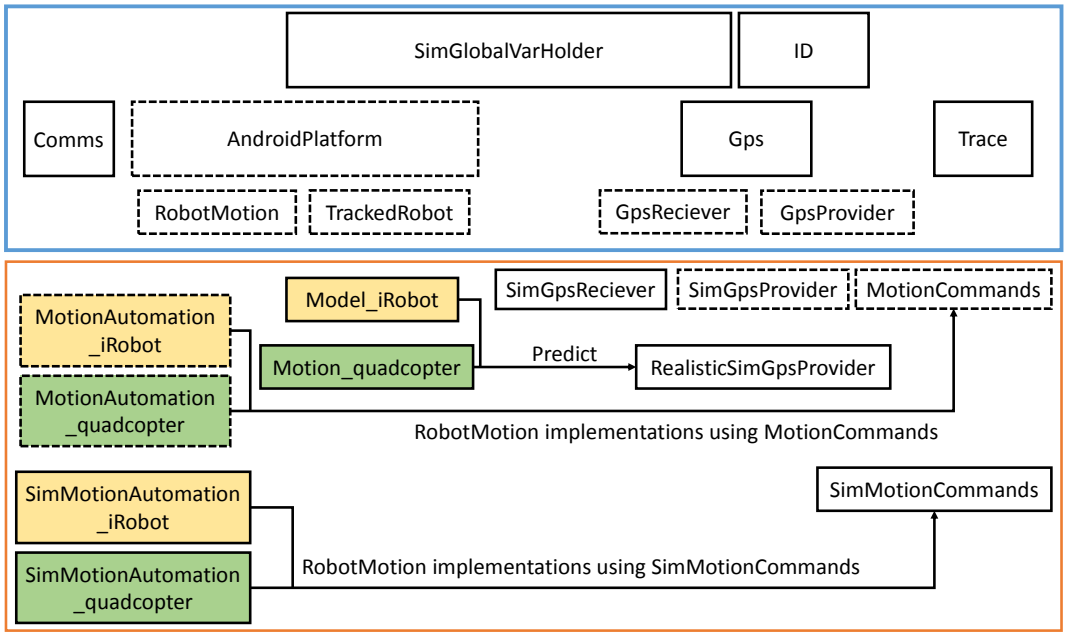

Figure 4.4: Heterogeneous robots in the StarL simulator

Figure 4.4 shows how the StarL simulator handles the motion of hetero-

geneous robots. Files in the top blue box are in the interface layer and the files that provide implementation are in the orange box on the bottom. Java interfaces and abstract classes are shown in rectangles with dashed lines. The yellow-filled boxes are iRobot Create platform-specific implementations while the green-filled boxes are AR DRONE2 platform-specific implementations. The RobotMotion and TrackedRobot are interfaces for supporting heterogeneous robots within one hardware platform (the simulated hardware platform). Other abstract classes and interfaces are written for accommodating StarL onto different hardware platforms.

CHAPTER 5

EXPERIMENTS AND RESULTS

In this chapter, we introduce a few demo StarL applications. Each application demonstrates the use of some StarL primitives. The experiment results are obtained using the StarL simulator.[1]

5.1 Intersection Coordination Protocol Application

Consider a four-way, double-lane, intersection that will be navigated by autonomous robotic vehicles through communication (see Figure 5.1). Each vehicle arrives at one of the *arrival zones* $A0, B0, C0, D0$ with a designated *departure zone* $A1, B1, C1, D1$. It coordinates with other vehicles according to an *intersection coordination protocol (ICP)* and proceeds to move through a sequence of *critical zones* A, B, C, D following certain right-hand traffic rules (e.g., no backing or U-turns). For example, a vehicle with source destination pair $(A0, D1)$ will have the path $A0, A, C, D, D1$. The requirements from the system are:

- (traffic_safety) No two vehicles occupy the same critical zone at the same time.
- (traffic_progress) There exists a timebound t within which every approaching vehicle departs.

We would also like the protocol to permit concurrent safe traversals. For examples, the vehicle with path $A0, A, A1$ should not block the vehicle with path $D0, D, B, B1$.

[1]This chapter includes previously published material [10, 11, 12].

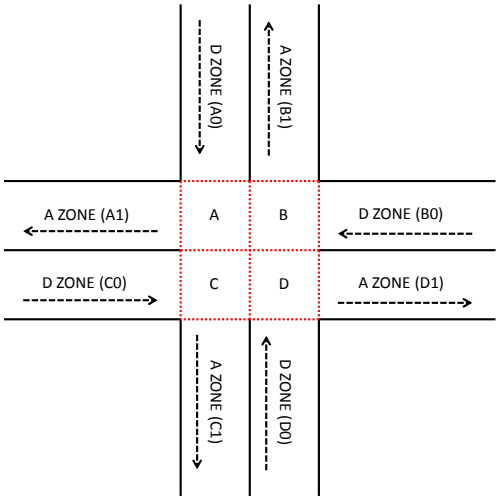

Figure 5.1: Four-way automatic intersection

5.1.1 Intersection Coordination Using StarL

A protocol for intersection works as follows: the participating vehicles agree on the set of participants; then they request access to the sequence of zones needed for traversal in the intersection from the set of agreed-upon participants; once they have access to the entire sequence, they start traversing; after a zone is crossed by a vehicle, the zone is released. In this presentation we assume that processes do not fail and robots do not get stuck.

Figure 5.2 shows the pseudo code for ICP highlighting the use of StarL primitives. Each vehicle participating in the coordination runs an instance of this protocol with the same identifier *0* that uniquely identifies the intersection. The local variable $myseq_i$ is a list of zones; it is initialized to the sequence of zones that i must traverse to go from its current position to its destination. *getpos*() is a StarL function which will return the vehicle position, updated by the location sensors. This protocol uses two StarL primitives: *Registration* and *Mutex*. Recall that *Registration* allows a set of processes in a neighborhood to agree on a subset that contains participating processes; and *Mutex* allows mutually exclusive access to one or a set of shared resources. *reg* and *mux* are instances of these primitives with the identifier *0*. Finally, the *loc* variable, of the enumerated type, is initialized to the value `S0`.

The protocol initiates by invoking the *do_register*() function to start the registration process, then *loc* is set to `Reg_wait`. When the registration pro-

cess returns successfully, the StarL variable *reg.rList* is set to a non-null value. From `Reg_wait`, the process moves to `Mux_wait` only if registration completes (*reg.rList* nonempty), and in that case *do_mutex* is invoked to obtain exclusive access to the sequence of zones $mid(myseq_i)$ (except the first and the last) from the processes in *reg.rList*. When the mutex process returns successfully, the StarL variable *mux.crit* is set to `true`. From `Mux_wait`, the process moves to `Mov_wait` only if mutex returns successfully (*mux.crit* true) and in that case *do_move* is invoked which sends from *myseq* a sequence of points to *MotionControl*. In `Mov_wait`, when the vehicle traverses the zone $myseq[0]$ and reaches $myseq[1]$, the zone $myseq[0]$ is removed from the list and *release*(`myseq[0]`) is called to release that zone to the mutual exclusion. When the vehicle i reaches its destination zone, the *loc* is changed to `S1`.

In Figure 5.2, line 1 creates a variable (*myseq*) to hold the list of needed zones. It is initialized by computing the sequence of critical zones plus the departure zone the vehicle needs to go through this intersection. Line 4 creates an instance of the registration primitive using the intersection ID. Similarly, an instance of the mutual exclusion primitive is created in the next line. When access is granted by *Mutex*, at line 17, it sends the motion command *do_ReachAvoid* and changes *loc* to *move_wait*. The doneFlag is raised if the robot has reached the vicinity of the destination at line 21. Then, line 23 releases(*release*) the previous critical zone. Line 26 moves (*do_ReachAvoid*) to the next critical zone in *myseq*. Additionally, location is changed to S_1 if there are no more zones in *myseq*. Line 26 waits until the vehicle has reached the vicinity of the departure zone, then the last critical zone is released and the *unRegister* method is called.

5.1.2 ICP Simulation Results

Creating the simulation in StarL is simple. Using our simulation template (Main.java), one needs to specify the application (Figure 5.2) to simulate along with some simulation parameters. For example, one can simulate the ICP with four robots, one hundred milliseconds average message delay with some obstacles in the physical environment, shown in Figure 5.3. One can also customize the visualizer to display some extra application-specific infor-

```
1  myseq: List[Zones] = path(getpos(),dest)
2  loc: enum {Reg_wait,Mux_wait,Mov_wait,S1};
3
4  reg = Registration(0);
5  mux = Mutex(0);
6
7  init()
8    eff reg.do_register(); loc = Reg_wait;
9
10 do_mutex()
11   pre loc == Reg_wait && reg.rList != null
12   eff mux.do_mutex(mid(myseq), reg.rList);
13     loc = Mux_wait;
14
15 od_mutex()
16   pre loc == Mux_wait && mux.crit(mid(myseq))
17   eff do_ReachAvoid(myseq[1]);
18     loc = Mov_wait;
19
20 release(s)
21   pre loc == Mov_wait && s == myseq[0] && doneFlag
22   eff mux.release(myseq[0]);
23     myseq = remove(myseq,0);
24
25 done()
26   pre loc == Mov_wait && myseq == dest
27   eff loc = S1;
```

Figure 5.2: Pseudo code of the ICP

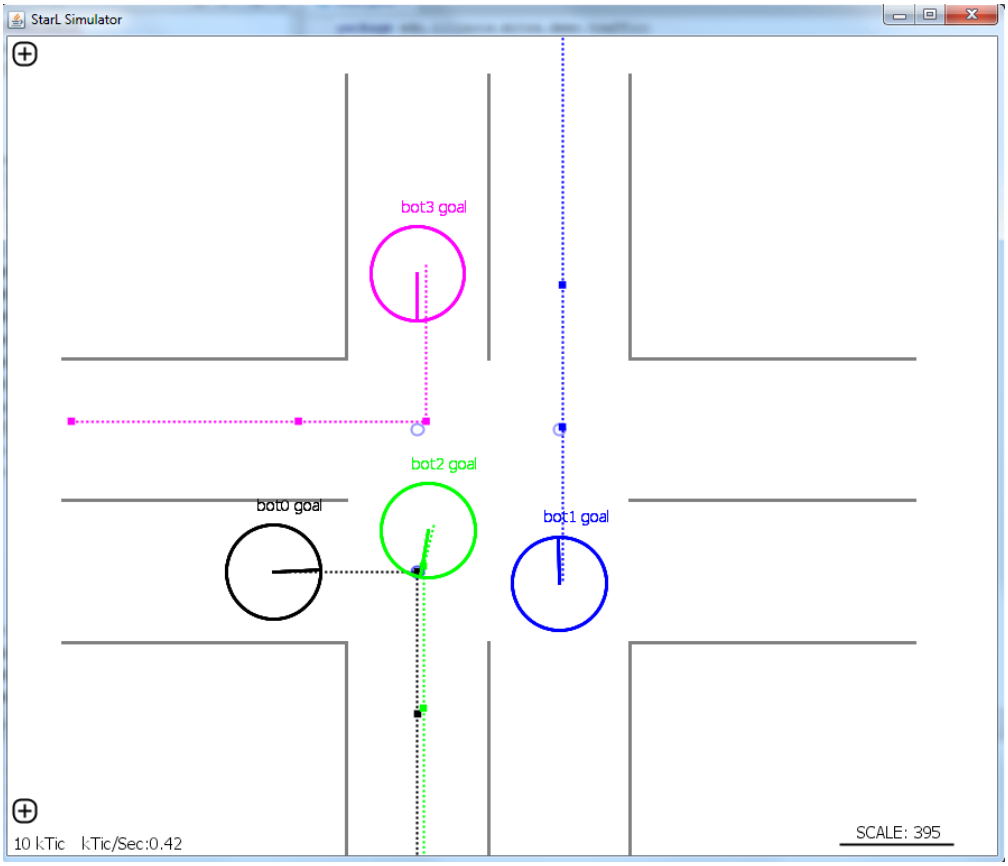

Figure 5.3: Screenshot of the ICP simulation

mation, such as the intended path for each robot.

A screenshot of the ICP simulation using four robots is shown in Figure 5.3. Robot2 starts in B0 (zones are defined in Figure 5.1), and intends to turn left. Robot1 starts in D0 and intends to go straight. Robot0, starting at C0, and Robot3, starting in A0, both intend to turn right. The dotted lines are intended zone sequence. Robot1 and Robot2 are in the intersection concurrently since their sets of critical zones are disjoint.

5.2 Distributed Search Application

In the distributed search application, a group of robots are given a house where they would locate an item. The house walls are defined as obstacles. Each obstacle is represented as a set of convex points. The entrance of each room defines the room and is given to the robots. The robots first use the *LeaderElection* primitive to elect a leader. Then the leader assigns a list

of rooms to each robot. Each robot goes to its assigned room and searches for the item using the room coverage algorithm. If no item is found in the current room, the robot move on to its next assigned room. If the item is found, the robot informs the group and the group stops searching.

5.2.1 Distributed Search Application Simulation Results

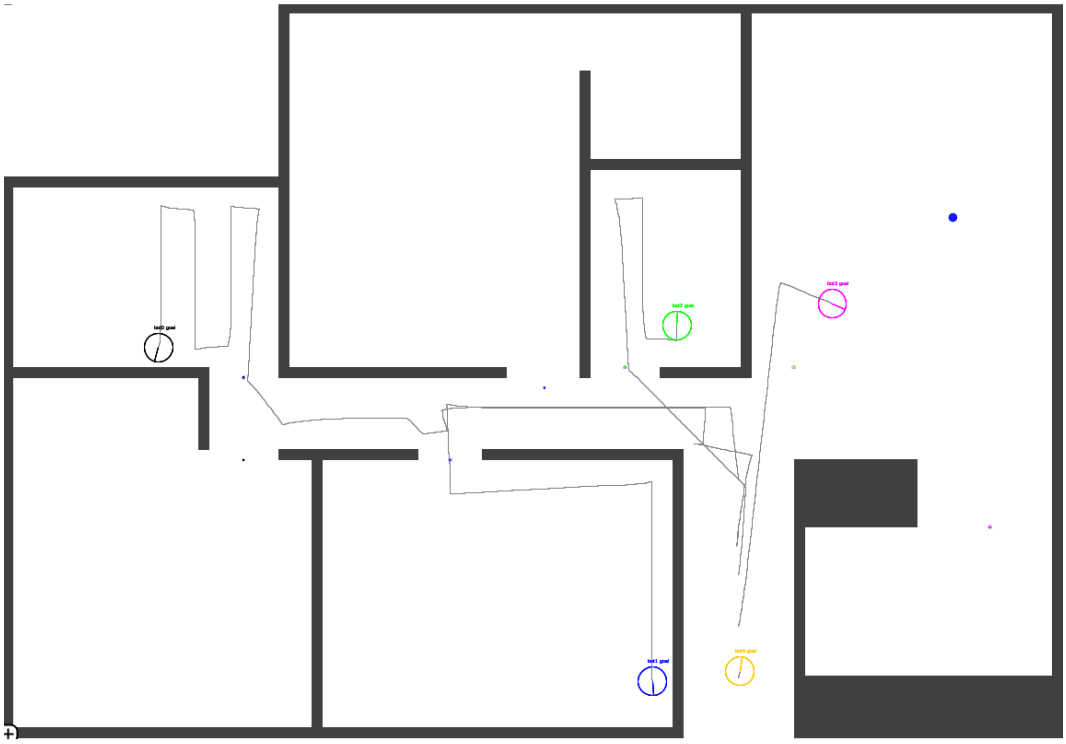

Figure 5.4: Screenshot of the Distributed Search application simulation

A simulation screenshot is shown in Figure 5.4. The item to be found is at the top-right corner shown in a blue circle. Thin gray lines are the robots' movement traces. The first three robots have entered their assigned rooms and started searching, the pink robot is moving toward its assigned room at the bottom-right corner. The yellow robot is still waiting for its assignment from the leader.

```
1  pid: Int = getId();
2  loc: enum{init, calc, wait} = init;
3  counter: Int = 0;
4  target: ItemPosition;
5  sharedsw pos[pid]: ItemPosition;
6
7  initialize()
8    pre loc == init
9    eff pos[pid] = getPos(); loc = calc;
10
11 update()
12   pre loc == calc ||!activeFlag
13   eff target = bisector(pos,pid,len);
14     doReachAvoid(target,getObstacles());
15     counter = 0; loc = wait;
16
17 wait()
18   pre loc == wait && counter <= 5
19   eff pos[pid] = getPos();
20     counter++;
21     if counter > 5 then loc = calc
```

Figure 5.5: Pseudo code of the Formation application

5.3 Formation Application

The simple formation application in Figure 5.5 illustrates the usage of the *DistributedSharedMemory* primitive. The application encodes a heuristic which when executed by an (odd) number of robots forms a regular polygon.

The *sharedsw* keyword declares $pos[i]$ as a single-writer (SW) multi-reader shared variable array. All the components of array $pos[]$ can be read by all all robots, but only robot i can write to $pos[i]$.

The formation application has three blocks. The *initialize*() block is executed once at the beginning and it sets $pos[i]$ to be the current position of the robot using the built-in *getPos* function. It also sets $loc = calc$ which ensures that only the *update*() block can execute next. The next block *update*() is enabled when $loc = calc$ or the *ReachAvoid* Boolean variable *activeFlag* is set to false. It computes a target position for the robot using the *bisector* function (not shown here). When executed by robot i, the *bisector* function

computes a point (at distance *len*) on the perpendicular bisector of the line joining $pos[j]$ and $pos[(j+1)\%n]$, where j and $(j+1)\%n$ are the robots diametrically opposite to i. It is straightforward to check that if $pos[]$ array forms a regular n-sided polygon then the application reaches an equilibrium. The *ReachAvoid* primitive is used to instruct the robot to move to the newly calculated target while avoiding static obstacles in the environments. Finally, the *wait*() block merely updates the $pos[i]$ variable and the *counter* so that the *wait* block is executed five times after every execution of *update*.

Two simulation screenshot are shown in Figure 5.6. Three robots, including one quadcopter and two iRobots, start in arbitrary locations. Because the RobotMotion interface is implemented differently for quadcopters and iRobots, the quadcopter needs to take off from the ground to reach a safe hight before moving to the point in space as shown in the top screenshot in Figure 5.6. Because the iRobot cannot move above the ground, we only run the *bisector* function in the x-y plane. If the robots only consist of quadcopters, the *bisector* function can be applied on $\mathbb{R}^3$. The robots converge quickly to a triangle as expected as shown in the bottom screenshot in Figure 5.6.

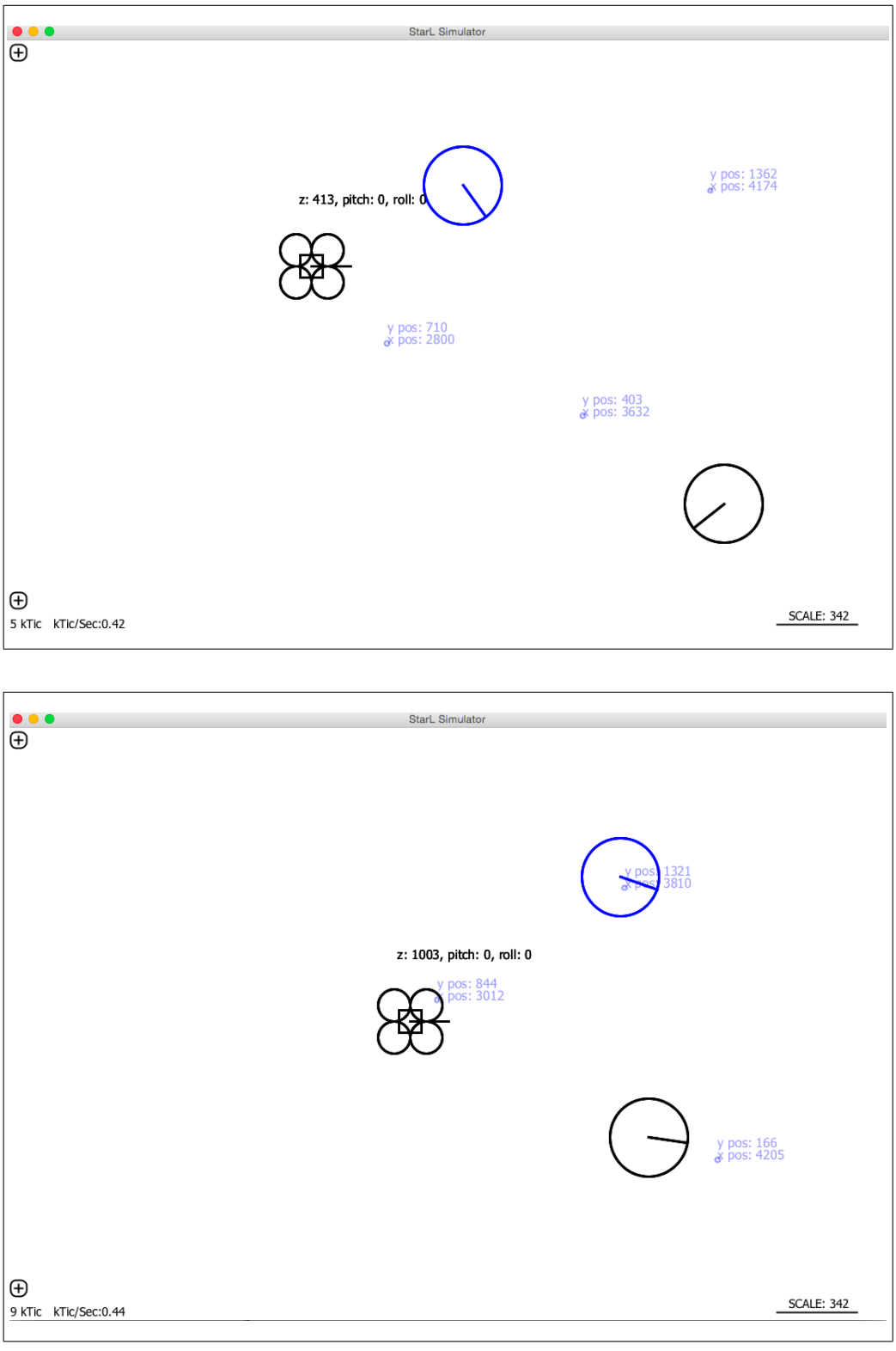

Figure 5.6: Screenshot of the Formation application simulation

CHAPTER 6

CONCLUSIONS

6.1 Conclusion

StarL framework simplifies the development of distributed robotic applications by providing programming abstractions and building blocks for the robot's motion, communication between robots and distributed coordination.

In this thesis, we introduced the programming abstractions as StarL primitives that are platform-independent and useful across hardware platforms, resulting in portability. We first introduced the primitives as building blocks to easily develop, simulate and debug distributed applications in StarL. Then, we discussed the design of the StarL framework which enables us to achieve portability of distributed robotic applications across hardware platforms. Thus, the same application, say, for formation control, can now be ported and deployed on multiple, heterogeneous robotic platforms. We also presented a few StarL applications and their simulation results.

6.2 Ongoing and Future Work

There are several directions for the future work on the StarL framework.

StarL High-Level Language The StarL high-level language provides a language that is at the level of abstraction of pseudo code. A preliminary complier is being developed by Ritwika Ghosh [11]. Continuation on this work would enable users who do not know Java to create their StarL applications.

StarL Web Interface The StarL web interface is being developed by Shuting Li [39]. The StarL web interface provides users with a cloud-based StarL

simulation environment, which saves the trouble for setting up the StarL development environments. It also provides an intuitive interface for users to create new StarL applications using the StarL high-level language. Simulation results will be displayed for easier debugging.

Sensors Generalization of sensors across different hardware platforms could be another big step for the StarL platform. Even though the motion has been generalized for a class of mobile robots, StarL currently has no programming abstraction for different sensors across different hardware platforms.

New Robot Models New robot models could be integrated to the StarL framework. New models can be used to simulated more complicated dynamics, and they can also bring a new set of StarL applications. For example, the model of a robot that consists of robotic arms could open up new research directions such as assembly line coordination.

Computational Resources for Each Robot Currently, the computational resources for each robot is not modeled in the StarL simulator. Implementing limited memory, limited computational power in the simulator could enable research in distributed systems that have limited computational resources [40].

Simulation Using Physics Engines Integrating the StarL framework with physics engines, such as Bullet, Open Dynamics Engine, and Gazebo, would allow realistic physics simulation to be generated.

Incorporate Linear Solver Libraries Integrating the StarL framework with linear solver libraries, such as JOptimizer, SCPSolver, and Java ILP, could allow more powerful path planning algorithms to be implemented. For example, a RRT path planing algorithm in higher dimensional space including the robot's speed and acceleration could be implemented.

REFERENCES

[1] D. M. Dilts, N. P. Boyd, and H. Whorms, "The evolution of control architectures for automated manufacturing systems," *Journal of Manufacturing Systems*, vol. 10, no. 1, pp. 79–93, 1991.

[2] N. Correll and A. Martinoli, "Multirobot inspection of industrial machinery," *IEEE Robotics Automation Magazine*, vol. 16, no. 1, pp. 103–112, March 2009.

[3] M. M. Waldrop, "Autonomous vehicles: No drivers required," *Nature News*, February 2015.

[4] C. Steiner, "Bot in the delivery:kiva systems," *Forbes Magazine*, March 2009.

[5] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Magazine*, vol. 29, no. 1, pp. 9–20, 2008. [Online]. Available: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2082

[6] J. L. Burke, R. R. Murphy, M. D. Coovert, and D. L. Riddle, "Moonlight in Miami: Field study of human-robot interaction in the context of an urban search and rescue disaster response training exercise," *Human–Computer Interaction*, vol. 19, no. 1-2, pp. 85–116, 2004.

[7] A. Bertozzi, M. Kemp, and D. Marthaler, "Determining environmental boundaries: Asynchronous communication and physical scales," in *Cooperative Control*, 2004.

[8] A. Zimmerman, "StarL for programming reliable robotic networks," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign, 2013.

[9] P. S. Duggirala, T. T. Johnson, A. Zimmerman, and S. Mitra, "Static and dynamic analysis of timed distributed traces," in *Proceedings of the 33rd IEEE Real-Time Systems Symposium, RTSS*, 2012, pp. 173–182.

[10] Y. Lin and S. Mitra, "StarL: Towards a unified framework for programming, simulating and verifying distributed robotic systems," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2015, CD-ROM, Portland, OR, USA, June 18 - 19, 2015*, 2015. [Online]. Available: http://doi.acm.org/10.1145/2670529.2754966 pp. 9:1–9:10.

[11] Y. Lin, R. Ghosh, and S. Mitra, "StarL framework for programming, simulating and verifying distributed robotic applications," *Submitted for review of the ACM Transactions on Embedded Computing Systems (TECS)*, 2016, submitted for review.

[12] Y. Lin, S. Mitra, and S. Li, "Porting code across simple mobile robots," *Submitted for review of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, submitted for review.

[13] M. Mesbahi and M. Egerstedt, *Graph-Theoretic Methods in Multiagent Networks*. Princeton University Press, 2010.

[14] J. Cortes, S. Martinez, T. Karatas, and F. Bullo, "Coverage control for mobile sensing networks," in *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, vol. 2, 2002, pp. 1327–1332.

[15] D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, and B. Stewart, "Distributed multi-robot exploration and mapping," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1325–1339, 2006.

[16] M. B. Dias, R. M. Zlot, N. Kalra, and A. T. Stentz, "Market-based multirobot coordination: a survey and analysis," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1257–1270, July 2006.

[17] K. M. Lynch and M. T. Mason, "Stable pushing: Mechanics, controllability, and planning," *International Journal on Robotics Research*, vol. 16, no. 6, pp. 533–556, 1996.

[18] D. Rus, "Coordinated manipulation of objects in a plane," *Algorithmica*, vol. 19, no. 1, pp. 129–147, 1997.

[19] K. Konolige, C. Ortiz, R. Vincent, B. Morisset, A. Agno, M. Eriksen, D. Fox, B. Limketkai, J. Ko, B. Stewart et al., "Centibots: Very large scale distributed robotic teams," in *Building the Information Society*. Springer, 2004, pp. 761–761.

[20] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 3293–3298.

[21] L. Chaimowicz, N. Michael, and V. Kumar, "Controlling swarms of robots using interpolated implicit functions," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. IEEE, 2005, pp. 2487–2492.

[22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, 2009, p. 5.

[23] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, vol. 1, 2003, pp. 317–323.

[24] D. Blank, L. Meeden, and D. Kumar, "Python robotics: An environment for exploring robotics beyond legos," in *ACM SIGCSE Bulletin*, vol. 35, no. 1. ACM, 2003, pp. 317–321.

[25] P. Corke, "A robotics toolbox for MATLAB," *Robotics and Automation Magazine, IEEE*, vol. 3, no. 1, pp. 24–32, 1996.

[26] I. Nesnas, "Claraty: A collaborative software for advancing robotic technologies," in *Proc. of NASA Science and Technology Conference*, vol. 2, 2007.

[27] I. A. Nesnas, "The claraty project: Coping with hardware and software heterogeneity," in *Software Engineering for Experimental Robotics*. Springer, 2007, pp. 31–70.

[28] D. Calisi, A. Censi, L. Iocchi, and D. Nardi, "Openrdk: A modular framework for robotic software development," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 1872–1877.

[29] V. Vladimerou, A. Stubbs, J. Rubel, A. Fulford, and G. Dullerud, "A hovercraft testbed for decentralized and cooperative control," in *Proc. of American Control Conference*, 2004.

[30] A. Stubbs, V. Vladimerou, A. T. Fulford, D. King, J. Strick, and G. E. Dullerud, "Multivehicle systems control over networks: A hovercraft testbed for networked and decentralized control," *Control Systems, IEEE*, vol. 26, no. 3, pp. 56–69, 2006.

[31] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 212–237, 2003.

[32] S. Karaman and E. Frazzoli, "Sampling-based optimal motion planning for non-holonomic dynamical systems," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, May 2013, pp. 5041–5047.

[33] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453–455, Aug. 1974. [Online]. Available: http://doi.acm.org/10.1145/361082.361093

[34] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, Nov. 1974. [Online]. Available: http://doi.acm.org/10.1145/361179.361202

[35] J. Aspnes and M. Herlihy, "Fast randomized consensus using shared memory," *J. Algorithms*, vol. 11, no. 3, pp. 441–461, Sep. 1990. [Online]. Available: http://dx.doi.org/10.1016/0196-6774(90)90021-6

[36] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, pp. 40–44, 2009.

[37] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, 2011.

[38] N. Hervey, "Localization and control of distributed mobile robots with the Microsoft Kinect and StarL," M.S. thesis, University of Texas at Arlington, Texas, 2016.

[39] S. Li, "Toward a web interface for distributed robotics," B.S. thesis, University of Illinois at Urbana Champaign, 2016.

[40] A. Cornejo, A. Dornhaus, N. Lynch, and R. Nagpal, "Task allocation in ant colonies," *Distributed Computing: 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pp. 46–60, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-45174-8_4